Sekvenser

I. LITT AV ADT-HIERARKI

I.1 CONTAINER

I.2 ITERATOR – ENUMERATION ADT

II. RANKEDSEQUENCE ADT

III. POSISJON & POSITIONALSEQUENCE ADT

III.1 SEQUENCE ADT

IV. IMPLEMENTASJON

IV.1 ARRAY IMPLEMENTASJON

IV.2 LISTE IMPLEMENTASJON

V. GENERISK SORTERING AV SEKVENSER

Kap	unntatt	kursorisk
3	3.2.4, 3.5	3.1.3, 3.2.3
4		4.4

+ implementasjon av en ADT med en annen ADT)

Stabel og Kø ADT

package jdsl.simple.api;

```
/** LIFO-kø: S.push(o).peek() = o
 */
public interface Stack {
/** legger nye Objekter på toppen av stabel
 * @param o      Objektet som skal settes inn */
    void push(Object o);

/** fjerner top Objektet fra stabel
 * @return       top (=sist innsatte) Objektet
 * @exception    NullPointerException  hvis empty()*/
    Object pop();

/** returnerer (uten å fjerne) top Objektet fra stabel
 * @return       top (=sist innsatte) Objektet
 * @exception    NullPointerException  hvis empty() */
    Object top();

/** @return      true hviss stabel er tom */
    boolean isEmpty();

/** @return      antall elementer i Køen */
    int size();

}
```

```
/**  FIFO-kø:
 *   if (!S.isEmpty()) S.enqueue(o).front() = S.front()*/
public interface Queue {
/**  legger nye Objekter i enden av Kø
 *  @param o     Objektet som skal settes inn */
     void enqueue(Object o);

/**  fjerner første Objektet fra Kø
 *  @return      front (=først innsatte) Objektet
 *  @exception   NullPointerException  hvis isEmpty() */
     Object dequeue();

/**  returnerer (uten å fjerne) første Objektet fra Kø
 *  @return      front (=først innsatte) Objektet
 *  @exception NullPointerException  hvis isEmpty() */
     Object front();

/**  @return     true hvis Køen er tom */
    boolean isEmpty();

/**  @return     antall elementer i Køen */
    int size();
}
```

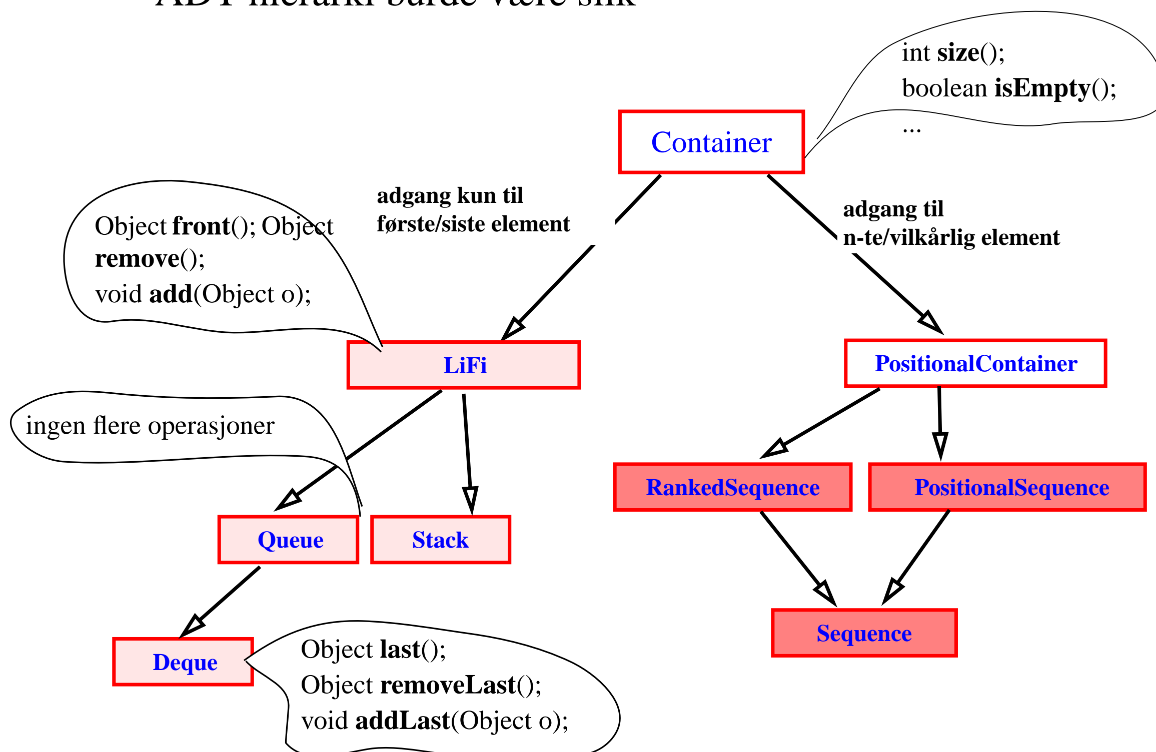
flere implementasjoner ligger i : **jdsl.simple.ref**

Container ADT

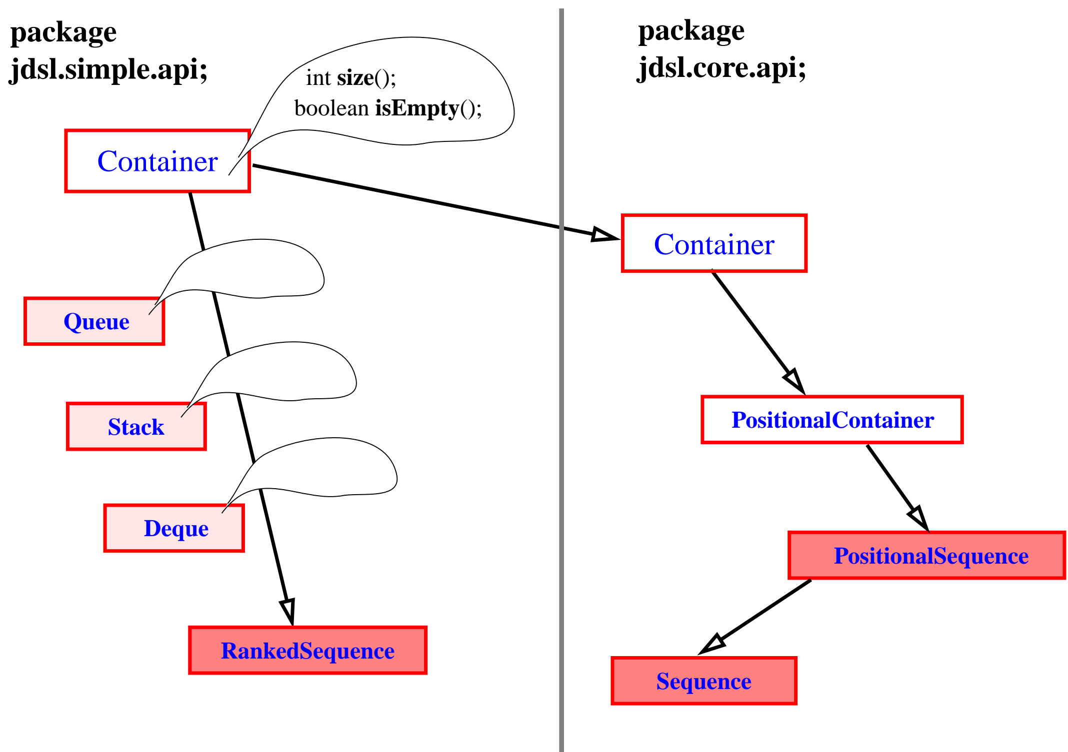
package jdsl.simple.api;

```
/**  generisk samling av Objekter
 *   supertype for alle samlinger
 */

public interface Container {

/**  @return    true hviss stabel er tom */
     boolean isEmpty();

/**  @return    antall elementer i samlingen */
     int size();
}
```

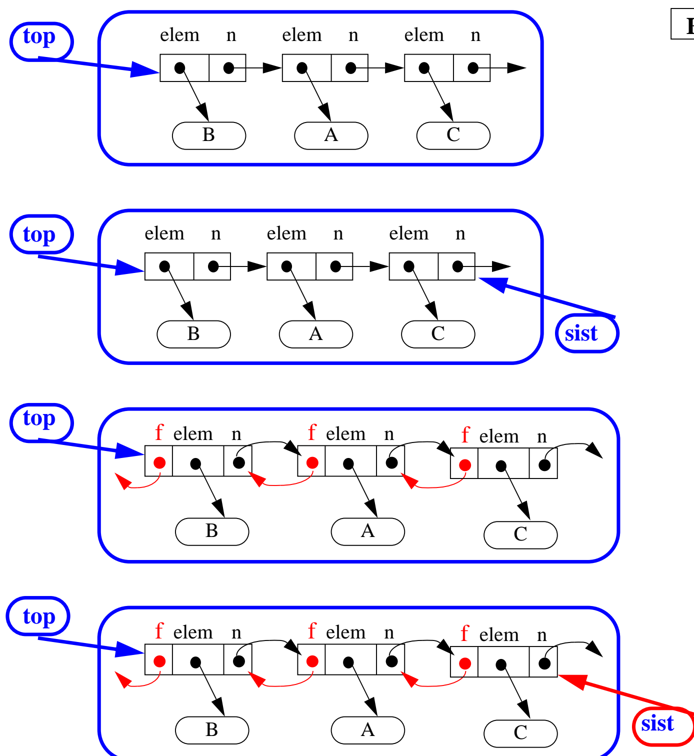

ADT hierarki burde være slik



... men det er slik

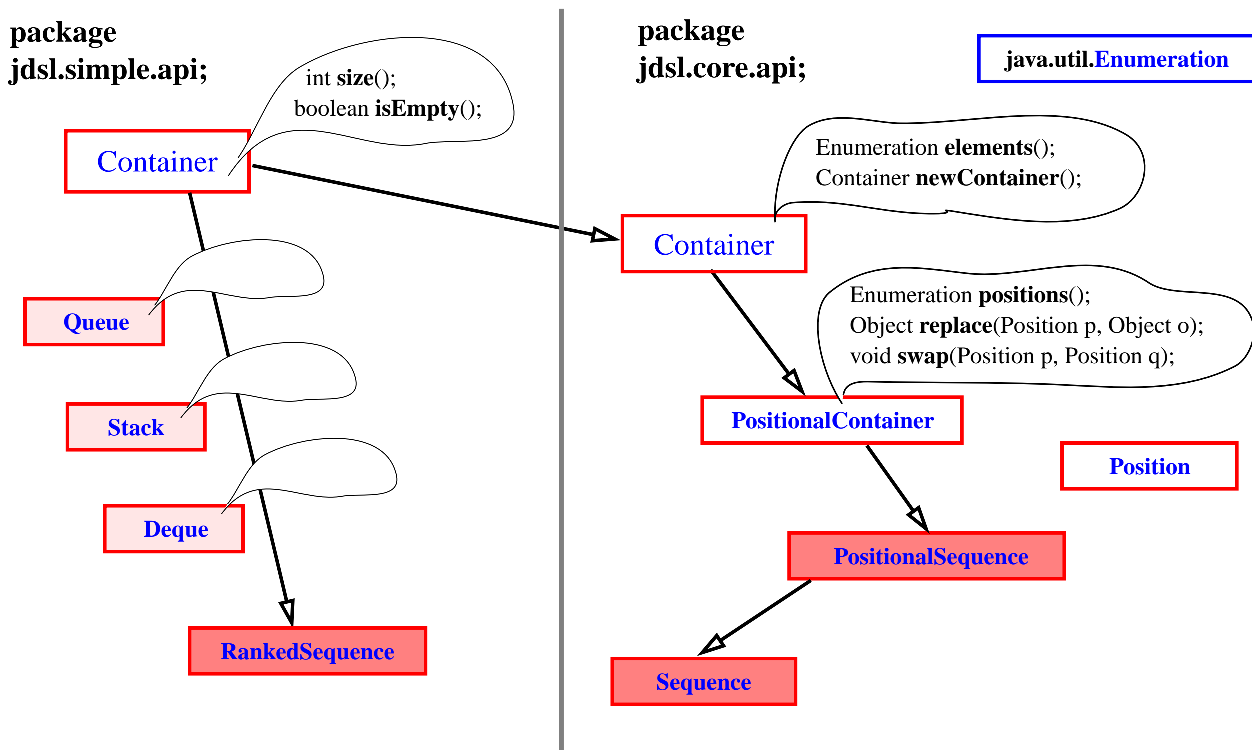


Listebaserte Data Strukturer



travers.	fra	innsetting	fjerning
EN-VEIS			
neste	første	første	første (Stack)
neste	første, siste	første siste	første (Queue)
TO-VEIS			
neste, forrige	første	første vilkårlig	første vilkårlig
neste, forrige	første, siste	første, vilkårlig, siste	første, vilkårlig, siste (Deque)

... men det er altså slik



Container og Enumeration ADT

package jdsl.core.api;

```
import java.util.Enumeration
/** generisk samling av Objekter */
public interface Container extends jdsl.simple.Container {
/** @return     enumerering av alle elementene fra samlingen */
      Enumeration elements();
/** @return     ny samling
               av samme type */
      Container newContainer();
}
```

package java.util;

```
/** for å traversere en samling en gang */
public interface Enumeration {
/**  @return true hviss flere elementer */
     boolean hasMoreElements();
/**  @return  neste element
 *            første kall gir første elementet
 *   @exception NoSuchElementException
 *             hvis ikke flere elementer */
     Object nextElement();        }
```

f.eks. for å skrive ut alle elementene fra en

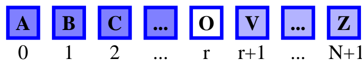
```
Container C;
Enumeration en = C.elements();
while (en.hasMoreElements())
      System.out.println(en.nextElement());
```

implementasjon med **array** *og* **java.util.Vector** *finnes i* **package jdsl.core.ref***;*

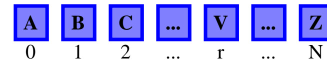
RankedSequence ADT

package jdsl.simple.api;

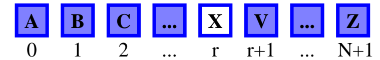
```
/** adgang til/fjerning/innsetting av elementer
 *  i vilkårlig posisjon (ulik Stack og Queue)
 *  Posisjon – 'rank' – angis med et tall 0 <= r < size() */

public interface RankedSequence
        extends (jdsl.simple.)Container {

/** setter inn et nytt Objekt o i posisjon r
 *  @param       r  et tall: 0 <= r <= size()
 *  @param       o  Objektet som skal innettes ved r
 *  @exception   IllegalPosition hvis r < 0 eller r > size()
 */ void insertElemAtRank(int r, Object o);
```



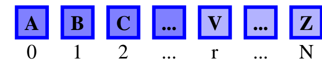

```
/** returnerer Objektet i posisjon r
 *  @param       r  et tall: 0 <= r < size()
 *  @return      Objektet  i posisjon r
 *  @exception   IllegalPosition hvis
 *               r < 0 eller r > size()-1
 */ Object elemAtRank(int r);
```




```
/** erstatter Objektet i posisjon r med et nytt Objekt
 *  @param       r  et tall: 0 <= r < size()
 *  @param       o Objektet som skal innettes
 *  @return      Objektet  som ble erstattet ved r
 *  @exception   IllegalPosition
 *               hvis r < 0 eller r > size()-1
 */ Object replaceElemAtRank(int r, Object o);
```




```
/** fjerner og returnerer Objektet i posisjon r
 *  @param       r  et tall: 0 <= r < size()
 *  @return      Objektet  som ble fjernet ved r
 *  @exception   IllegalPosition
 *               hvis r < 0 eller r > size()-1
 */ Object removeElemAtRank(int r);
}
```



Implementasjon av en ADT med ... en annen ADT !

Deque med RankedSequence

1. Data Representasjon:

A B C ... **Z** = [A] [B] [C] [...] [Z]
0 1 2 ... **size()-1**

2. Data Struktur:

= **RankedSequence** r

3. Data Invariant:

= ***ingen*** (enhver RankedSequence representerer en Deque)

4. Implementasjon:

first()	=	r.**elemAtRank**(0)
last()	=	r.**elemAtRank**(r.**size**() –1)
size()	=	r.**size**()
insertFirst(e)	=	r.**insertElemAtRank**(0,e)
insertLast(e)	=	r.**insertElemAtRank**(r.**size**(),e)
removeFirst()	=	r.**removeElemAtRank**(0)
removeLast()	=	r.**removeElemAtRank**(r.**size**() – 1)

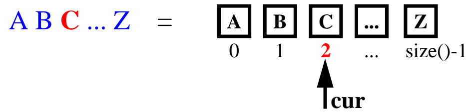
```
class DQrs implements Deque {
  private RankedSequence r;
  public DQrs(RankSequence a)  { r = a ; }
  ...
}
```

Enhver implementasjon av RankedSequence *sent som aktuell parameter til* DQrs *vil gi oss en implementasjon av* Deque

Implementasjon av en ADT med en annen ADT

EditorLine med **RankedSequence**

1. Data Representasjon:



2. Data Struktur:

= **RankedSequence** r
int cur

3. Data Invariant:

= **0 <= cur < r.size()**
alle objektene i r er av type
(kan omstøpes til) char

4. Implementasjon:

```
class ELrs implements EditorLine {
  private RankedSequence r;
  private int cur;
  public ELrs(RankedSequence a) { r = a ; }
  ...
```

```
moveLeft()   =  if (cur > 0) cur--;
moveRight()  =  if (cur < r.size()–1) cur++;
moveEnd()    =  cur= size()–1;
moveStart()  =  cur= 0;
insert(c)    =  if (DI()) r.insertElemAtRank(
                    cur++, new Character(c));
replace(c)   =  if (DI()) r.replaceElemAtRank(
                    cur, new Character(c));
delete()     =  if (DI()) r.removeElemAtRank(cur);
charAt(i)    =  if (DI(i))
                    Object o = r.elemAtRank(i)
                    return ((Charater)o).charValue();
find(c)      =  int h= cur;
                for (int i=cur; i< r.size(); i++)
                    if (charAt(i) == c) cur=i; break;
                if (cur<size()) return cur; else return h;
DI(i)        =  return (0 <= i && i < r.size())
DI()         =  return DI(cur)
```

Enhver implementasjon av RankedSequence *sent som aktuell parameter til* ELrs *vil gi oss en implementasjon av* EditorLine

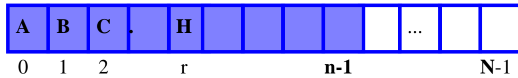
Implementasjon av RankedSequence

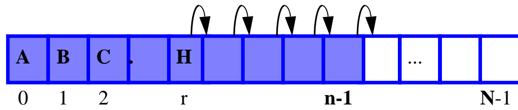
med ***java.util.Vector***

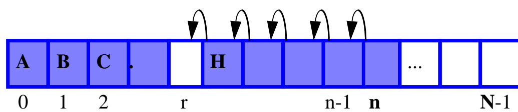
```
package java.util;

public class Vector {
...
void ensureCapacity(
      int minCapacity) { ... }
void insertElementAt(
      Object o, int i) { ... }
void setElementAt(
      Object o, int i) { ... }
void removeElementAt(
      int i) { ... }
Object elementAt(int i) { ... }
Object lastElement() { ... }
Object firstElement() { ... }
int capacity() { ... }
int size() { ... }
}
```

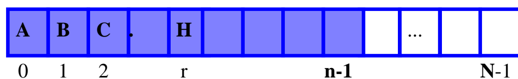
```
import java.util.Vector;
public class RSvector implements RankedSequence {
  private Vector v;
  public RSvector() { v= new Vector(); }
  public void insertElemAtRank(int r, Object o)
        throws IllegalPosition {
    try{ v.insertElementAt(o, r); }  // sjekker rank r
    catch(ArrayIndexOutOfBoundsException e) {
        throw new IllegalPosition(); } }
  public void elemAtRank(int r) {  // sjekk rank r
    return v.elementAt(r);  }
  public Object removeElemAtRank(int r) { //sjekk rank r
    Object o = v.elementAt(r);
    v.removeElementAt(r); return o;  }
  public Object replaceElemAtRank(int r) { //sjekk rank r
    Object o = v.elementAt(r);
    v.setElementAt(r); return o;  }
  public int size() { return v.size();  }
  public boolean isEmpty() { return v.size() == 0; }
}
```

Implementasjon av RankedSequence *med* ***array***










```
private Object[] A ; private int n ;
int size() { return n; }                          O(1)
bolean isEmpty() { return n==0 }                  O(1)
Object ElemAtRank(r) {                            O(1)
   return A[r] }
Object replaceElemAtRank(r, E) {                  O(1)
   o = A[r];   A[r] = E
   return o }
void insertElemAtRank(r, E) {                     O(n)
   for i = n-1, n-2, ... r
         A[i+1] = A[i]
   A[r] = E
   n = n+1
}

Object removeElemAtRank(r) {                      O(n)
   o = A[r]
   for i = r, r+1, ... n-1
         A[i] = A[i+1]
   n = n-1
   return o;
}
```

meget effektive oppslag
men mindre effektiv innsetting / fjerning

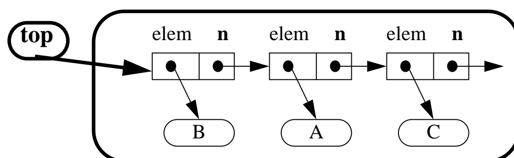
Implementasjon av RankedSequence

med ***en-veis liste***

DataStruktur

private Node top ;

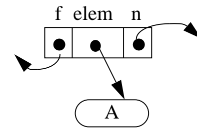
private int n; // antall elementer




```
// hjelpemetode
private Node nodeAtRank(int r) {
   Node c = top; int j = 0;
   while (c != null && j<r) {
      c = c.getNext(); j++; }
   return c;
}                                                 O(n)
```

```
public Object elemAtRank(int r)
      throws IllegalPosition {
   Node c = nodeAtRank(r);
   if (c == null)
      throw new IllegalPosition("!!!");
   else return c.element();
}                                                 O(n)

insertElemAtRank(int r, Object E)
         throws IllegalPosition  {
   if (r>n || r<0) throw new IllegalPosition("");
   if (r==0) top = new Node(E,top);
   else {
     DLNode forrige= nodeAtRank(r–1);
     DLNode ny = new DLNode(E,forrige.getNext());
     forrige.setNext(ny);
     }
   n++;
}                                                 O(n)
 . . .
```

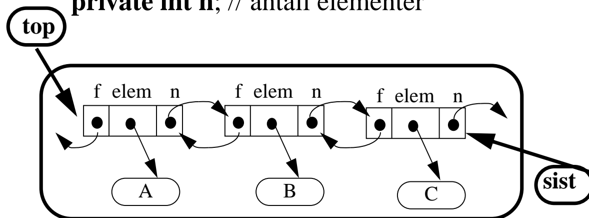
RankedSequence *med* ***to-veis liste***




```
public class DLNode
{  private DLNode f, n;
   private Object elem;
   public DLNode(Object o, DLNode ff, DLNode nn) { elem= o; n= nn; f= ff; }
   public Object element() { return elem; }
   public DLNode getNext() { return n; }
   public DLNode getPrev() { return f; }
   public void setElem(Object o) { elem= o; }
   public void setNext(DLNode d) { n= d; }
   public void setPrev(DLNode d) { f= d; }
}
```

DataStruktur

```
      private DLNode top, sist;
      private int n; // antall elementer
```



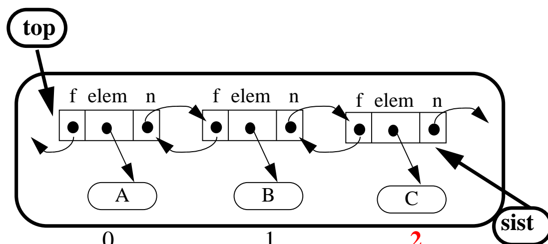

```
// hjelpemetode
private DLNode nodeAtRank(int r) {
   if (r<0 || r>n) return null;
   else if (r < n/2) {
      DLNode c = top; int j = 0;
      while (c != null && j<r) {
         c = c.getNext();
         j++; }
      return c;
   } else {
      DLNode c = sist; int j = n–1;
      while (c != null && j>r) {
         c = c.getPrev();
         j--; }
      return c;
} }
```

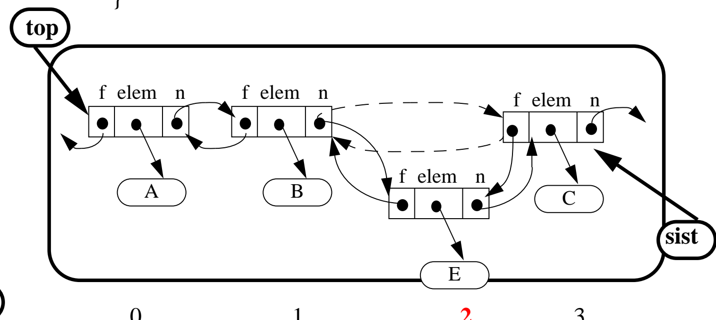
$n/2 = O(n)$

RankedSequence *med* ***to-veis liste***

```
insertElemAtRank(int r, Object E)
      throws IllegalPosition  {
   if (r>n || r<0) throw new IllegalPosition("“");
   DLNode neste= nodeAtRank(r);             —— O(n)
   DLNode forrige= neste.getPrev();         }
   DLNode ny = new                          }
         DLNode(E,neste,forrige);           } O(1)
   neste.setPrev(ny);                       }
   if (forrige != null) forrige.setNext(ny); }
   n++;                                     }
// ev. oppdater top/sist
}
```

```
removeElemAtRank(int r)
      throws IllegalPosition  {
   if (r>=n || r<0) throw new IllegalPosition("“");
   DLNode rem= nodeAtRank(r);                         —— O(n)
   if (rem.getNext() != null)                         }
      rem.getNext().setPrev( rem.getPrev() );         }
   if (rem.getPrev() != null)                         } O(1)
      rem.getPrev().setNext( rem.getNext() ) ;        }
   n--;                                               }
   return rem.element() ;                             }
// ev. oppd.top/sist
}
```





Implementasjoner av RankedSequence

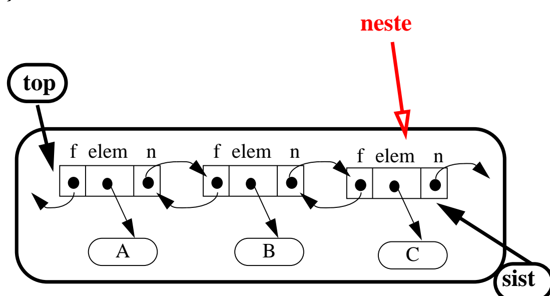
kompleksitet / operasjon	Vector *relativt!!!*	**Array**	**en-veis liste**	**to-veis liste**
Container				
size	*1*	1	1	1
isEmpty	*1*	1	1	1
RankedSequence				
elemAtRank	*1*	**1**	n	n
insertElemAtRank	*1*	n	n	n
removeElemAtRank	*1*	n	n	n
replaceElemAtRank	*1*	**1**	n	n

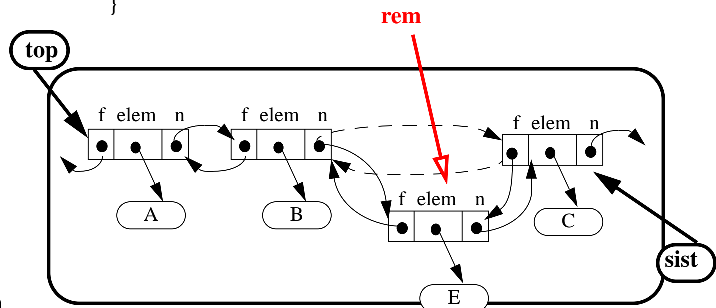
DL-Sequence

@param **neste** en node i listen
insertElemAt(DLNode neste, Object E)
~~throws IllegalPosition {~~
~~if (r>n || r<0) throw new IllegalPosition("");~~
~~DLNode neste= **nodeAtRank**(r);~~
DLNode forrige = neste.getPrev();
DLNode ny = new
DLNode(E,neste,forrige);
neste.setPrev(ny);
if (forrige != null) forrige.setNext(ny);
n++;
// ev. oppdater top/sist
}

$O(1)$

@param **rem** en node i listen
removeElemAt(DLNode rem)
~~throws IllegalPosition {~~
~~if (r>=n || r<0) throw new IllegalPosition("");~~
~~DLNode rem= **nodeAtRank**(r);~~
if (rem.getNext() != null)
rem.getNext().setPrev(rem.getPrev());
if (rem.getPrev() != null)
rem.getPrev().setNext(rem.getNext()) ;
return rem.element() ;
n--;
// ev. oppd.top/sist
}





DL-Sequence ... ADT?

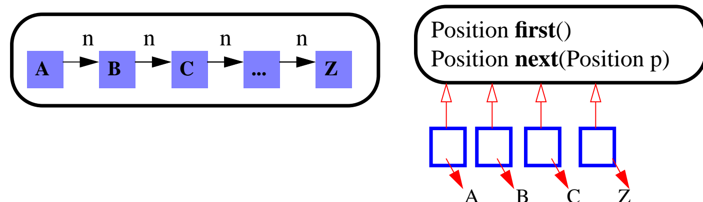
/**Sekvens der elementene aksesseres gjennom DLNode */

```
public interface DL-Sequence
            extends Container {
// Position aksess ______________________          implem.

/**   @return første noden i sekvensen
 *    første elem fåes ved first().getElem() */
   DLNode first();                                    O(1)

/**   @return første posisjon i sekvensen */
   DLNode last();                                     O(1)

/**   @param v en posisjon i sekvensen
 *    @return posisjon før v
 *    @exception IllegalPosition hvis v==first() */
   DLNode before(DLNode v);                           O(1)

/**   @param v en posisjon i sekvensen
 *    @return posisjon etter v
 *    @exception IllegalPosition hvis v==last() */
   DLNode after(DLNode v);                            O(1)

// element håndtering ______________________
/**   @return posisjon av det innsatte elementet */
   DLNode insertFirst(Object e);                      O(1)

/**   @return posisjon av det innsatte elementet */
   DLNode insertLast(Object e);                       O(1)

/**   @return posisjon av det innsatte elementet */
   DLNode insertBefore(DLNode v, Object e);           O(1)

/**   @return posisjon av det innsatte elementet */
   DLNode insertAfter(DLNode v, Object e);            O(1)

/**   @return Objektet som ble erstattet */
   Object replace(DLNode v, Object e);                O(1)

/**   @return Objektet som ble fjernet */
   Object remove(DLNode v);                           O(1)

/**   bytt om Objekter lagret i v og w*/
   void swap(DLNode v, DLNode w);                     O(1)
// int size();  boolean isEmpty();                    O(1)
}
```

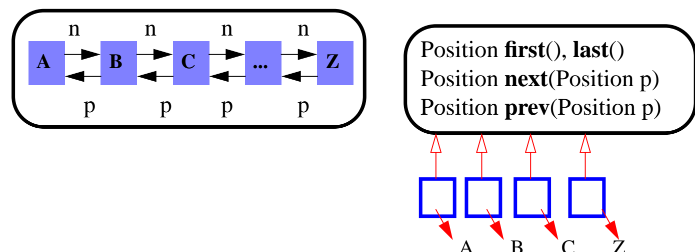
Dette er ikke et "riktig" interface fordi den krever en bestemt implementasjon med DLNode

Position ADT

= *abstraksjon av "et sted" som kan lagre noe*

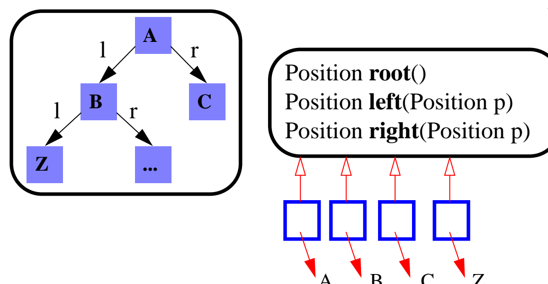



```
package jdsl.core.api;
public interface Position {
   Object element();
   Container container();
   void setElement(Object o);
}
```



"Rå struktur"

= aksess-metoder til en samling av Position*'er*



PositionalContainer ADT

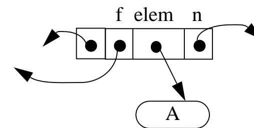
```
package jdsl.core.api;
 import java.util.Enumeration
 /**  generisk posisionell samling av Objekter  */
 public interface PositionalContainer extends jdsl.api.Container {
 /**  @return     enumerering av alle posisjoner fra samlingen */
      Enumeration positions();

 /**  bytt Objektene lagret ved argumentposisjoner
  *   @param      p, q posisjoner i samlingen
  *   @exception  InvalidPositionException hvis p eller q ikke er i denne samlingen */
      void swap(Position p, Position q) throws InvalidPositionException;

 /**  erstatt Objektet ved posisjon p med Objektet o
  *   @param      p posisjon der Objektet skal erstattes
  *   @param      o det nye Objektet som skal plasseres ved p
  *   @return     det gamle Objektet lagret opprinnelig ved p
  *   @exception  InvalidPositionException hvis p ikke er i denne samlingen */
      Object replace(Position p, Object o) throws InvalidPositionException;
// int size(); boolean isEmpty();   // Enumeration elements();  Container newContainer();
 }
```


PositionalSequence ADT

```
package jdsl.core.api;

public interface PositionalSequence
              extends PositionalContainer {
// Position aksess ________________________            implem.

/**   @return første posisjonen i sekvensen
 *    første elem fåes ved first().element() */
   Position first();                                    O(1)

/**   @return første posisjon i sekvensen */
   Position last();                                     O(1)

/**   @param v en posisjon i sekvensen
 *    @return posisjon før v
 *    @exception IllegalPosition hvis v==first() */
   Position before(Position v);                         O(1)

/**   @param v en posisjon i sekvensen
 *    @return posisjon etter v
 *    @exception IllegalPosition hvis v==last() */
   Position after(Position v);                          O(1)

// element håndtering ________________________

/**   @return posisjon av det innsatte elementet */
   Position insertFirst(Object e);                      O(1)

/**   @return posisjon av det innsatte elementet */
   Position insertLast(Object e);                       O(1)

/**   @return posisjon av det innsatte elementet */
   Position insertBefore(Position v, Object e);         O(1)

/**   @return posisjon av det innsatte elementet */
   Position insertAfter(Position v, Object e);          O(1)

/**   @return Objektet som ble fjernet */
   Object remove(Position v);                           O(1)

// int size();  boolean isEmpty();                      O(1)

// Enumeration elements();                              O(n)
// Container newContainer();                            O(1)

// Enumeration positions();                             O(n)
// void swap(Position v, Position w);                   O(1)
// Object replace(Position v, Object e);                O(1)
}
```

implementasjon med	*krever at*
en-/to-veis liste	Node implements Position
array	*"indeks"* implements Position

PositionalSequence *med* ***to-veis liste***




```
public class DLNode implements Position
{  private DLNode f, n;
   private Object elem;
   private Container sam;
   public DLNode(Object o, DLNode ff, DLNode nn, Container s)
   {   elem= o; f= ff; n= nn; sam= s; }
   public Object element() { return elem; }
   public void setElement(Object o)
       { elem= o; }
   public container() { return sam; }
   public DLNode getNext() { return n; }
   public DLNode getPrev() { return f; }
   public void setNext(DLNode d) { n= d; }
   public void setPrev(DLNode d) { f= d; }
}
```

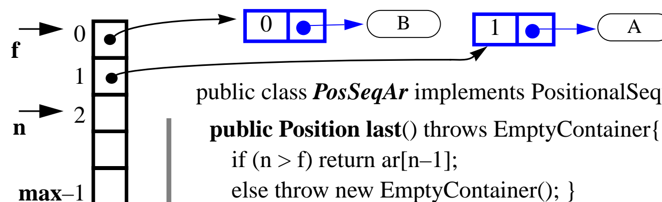
DataStruktur

```
        private DLNode top, sist;
   // antall elementer
        private int n;
```

```
public class PSdl implements PositionalSequence {
   public Position last() { return sist; }
   public Position before(Position p) {
      return ((DLNode)p).getPrev() ; }
   public Position insertFirst(Object o) {
      top= new DLNode(o, null, top, this) ;
      n++;
      return top ; }
   public Position insertBefore(Position p, Object o) {
      DLNode pp = (DLNode)p;
      DLNode ny = new DLNode(o, pp.getPrev(), pp, this);
      n++;
      return ny; }
   . . .
}
```


PositionalSequence *med* ***array***

Data Representasjon




```
public class ArPos
implements Position
{  private int r;
   private Object obj;
   private Container sam;
   public ArPos(
       int i, Object o, Container s)
   {   r= i; obj= o; sam= s; }
   public element() { return obj; }
   public rank() { return r; }
   public setElement(Object o) { obj= o; }
   public container() { return sam; }
   public setRank(int i) { r= i ; }
}
```

Data Struktur og Invariant

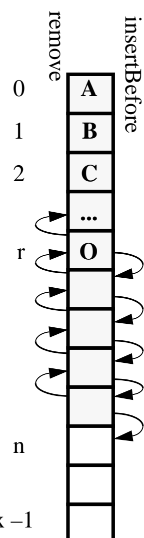
```
     private ArPos[N] ar;
     private int max; // størrelse til ar
     private int f, n; // første–neste ledig
```

f = 0, f <= n

isEmpty hviss f = n ; size = n

ArPos[f...n] er fylt med alle elementene

```
public class PosSeqAr implements PositionalSequence {
  public Position last() throws EmptyContainer{
     if (n > f) return ar[n–1];
     else throw new EmptyContainer(); }
  public Position before(Position p) throws InvalidPosExc
  {  if ( check(p).rank()==0) throw new  ...Exception();
     return ar[ check(p).rank()–1 ] ; }
  public Position insertLast(Object o) {
     ar[n] = new ArPos(n,o,this); n++; }
  public Position insertBefore(Position p, Object o)
               throws InvalidPositionExc {
     int r = check(p).rank();
     for (int k= n; k>r; k--) {
          ar[k]= ar[k–1]; ar[k].setRank(k); }
     ar[r]= new ArPos(r,o,this);
     n++ ;  return ar[r]; }    }
  private ArPos check(Position p) throws InvalidPositionExc {
     if (! p instanceOf ArPos) throw new InvalidPositionExc();
     ArPos a = (ArPos)p;
     if (p.rank() >= 0 && p.rank() <= n) return a;
     else throw new InvalidPositionExc();    }
```



Implementasjoner av PositionalSequence

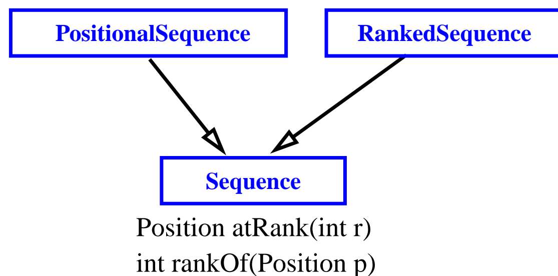
kompleksitet operasjon	Array	en-veis liste	to-veis liste
size, isEmpty	1	1	1
newContainer	1	1	1
elements, positions	n	n	n
replace, swap	1	1	1
first	1	1	1
last	1	n/1	1
before	**1**	**n**	**1**
after	1	1	1
insertFirst	n/1	1	1
insertLast	1	n/1	1
insertBefore	**n**	**n**	**1**
insertAfter	**n**	**1**	**1**
remove	**n**	**n**	**1**

Sequence ADT

size, isEmpty
elements, positions
newContainer
replace, swap

first, last
before, after
insertFirst/-Last
insertBefore/-After
remove

elemAtRank
replaceElemAtRank
insertElemAtRank
removeElemAtRank



Position atRank(int r)
int rankOf(Position p)

```
package jdsl.core.api ;
public interface Sequence
        extends PositionalSequence {
/** @param r rank
    @return posisjon ved rank r
    @exception BoundaryViolationException
  Position atRank(int r)
/** @param p posisjon
    @return rank av posisjonen p
    @exception InvalidPosition
  int rankOf(Position p)
/** @param r rank
    @param o objektet som skal settes inn
    @return posisjonen til det innsatte objektet
    @exception BoundaryViolationException
  Position insertAtRank(int r, Object o)
/** @param r rank
    @return objektet som ble fjernet
    @exception BoundaryViolationException
  Object removeAtRank(int rank)
}
```

Implementasjoner av Sequence

	kompleksitet operasjon	Array	en-veis liste	to-veis liste
Container	size, isEmpty	1	1	1
	newContainer	1	1	1
PositionalContainer	elements, positions	n	n	n
	replace, swap	1	1	1
PositionalSequence	first	1	1	1
	last	1	n/1	1
	before	1	n	1
	after	1	1	1
	insertFirst	n/1	1	1
	insertLast	1	n/1	1
	insertBefore	**n**	**n**	**1**
	insertAfter	**n**	**1**	**1**
	remove	**n**	**n**	**1**
Sequence	rankOf	1	1	1
	atRank	**1**	**n**	**n**
	insertAtRank	n	(n)	(n)
	removeAtRank	n	(n)	(n)

Sortering

- SelectionSort, MergeSort, ...
- *er en operasjon på strukturer med rekkefølgen på Posisjoner*

  total ordning: **for alle** Posisjoner p, q : p < q **eller** q < p **eller** q = p

  array, liste, RankedSequence, ...

  Sequence ADT *uttrykker denne totale ordningen med operasjoner before/after*
- *videre, må også elementene lagret i strukturen stå i en*

  total ordning: **for alle** Elementer a, b : a < b **eller** b < a **eller** a = b

Denne uttrykkes abstrakt (generelt) med

```
public interface Comparator
{       /** @return true iff a < b; false otherwise */
boolean isLessThan (Object a, Object b)
        /** @return true iff a > b; false otherwise */
boolean isGreaterThan (Object a, Object b)
        /** @return true iff a = b; false otherwise */
boolean isEqualTo (Object a, Object b)
        /** @return true iff a <= b; false otherwise */
boolean isLessThanOrEqualTo (Object a, Object b)
        /** @return true iff a >= b; false otherwise */
boolean isGreaterThanOrEqualTo (Object a, Object b)
}
```

forskjellige implementasjoner vil da definere forskjellige totale ordninger for forskjellige type objekter

Generisk seleksjon-sort av **Sequence**

```
for (k=0,1,2...< n) {    // n= S.size()
   int m = k;
   for (j= k+1...< n)                 1+2+...+(n-1)+n = O(n^2)
      if (S[j] < S[m]) m = j;
   S.swap( m, k )
}
```

en algoritme er generisk hviss alle klasse-parametre er ADTer : interface eller Object

```
void SS1(Sequence S, Comparator C)  {
   int k, j, min;
   int n = S.size() ;
   for (k=0; k<n; k++)  {
      min = k ;
      em = S.atRank(min).element();
      for (j= k+1; j<n; j++)
         if (C.isLessThan(
            S.atRank(j).element(), em ))
         {  min = j;
            em = S.atRank(min).element(); }
      S.swap( S.atRank(min), S.atRank(k) ) ;
}
```

kan brukes kun når **atRank**(**r**) *er* $O(1)$;

er den $O(n)$ *får vi SS1* $= O(n^3)$ *!!!*

```
void SS2(Sequence S, Comparator C)
   Position k, j, min;
   k= S.first();
   while ( k != S.last() )   {
      min = k;  j = k;
      while ( j != S.last() ) {
         j = S.after(j);
         if (C.isLessThan(j.element(), m.element()))
            min = j;
      }
      S.swap(min, k) ;
      k = S.after(k) ;
} }
```

first(), **last**(**r**), **after**(**p**) *kan alltid fåes* $O(1)$;

SS2 $= O(n^2)$ *!!!*

Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {        S[0...k-1] er sortert og
                           har k minste elem. fra S
   int m = k;
   for (j= k+1...< n) {    S[m] er minst i S[k...j-1]
      if (S[j] < S[m])
         m = j;
   }                       S[m] er minst i S[k...n–1]
   S.swap( m, k )
}                          S[0...n–1] er sortert
                           //og har n minste elem. fra S
```

```
// n= S.size() – 1
for (k=n...> 0) {          S[k+1...n] er sortert og har
                           n–k største elem. fra S

   for (j= 0...<k) {       S[j] er størst i S[0...j]
      if (S[j+1] < S[j])
         S.swap( j, j+1 )
   }                       S[k] er størst i S[0...k]

}                          S[1...n] er sortert
                           og har n største elem. fra S
```

```
void BubbleSort(Sequence S, Comparator C)  {
   Position p1, p2 ;
   for ( k = S.size()–1; k>0; k--)   {
      p1 = S.first() ;
      for ( j = 0; j<k; j++) {
         p2 = S.after(p);
         if (C.isLessThan(p2.element(), p1.element()))
            S.swap(p1,p2);
         p = a;
} } }
```

Korrekthet av boble-sortering

BubbleSort(int[] A) { // n er største indeks i tabellen

INN1: k=n : A[n+1..n] ...0 <= r < s <= n & s > n ...ingen slik s

1:***for (k = n, k > 0, k--)*** {

LI1: 0 <= r < s <= n & s > k → A[r] <= A[s]

dvs. A[k+1...n] er sortert og har n-k største elementer

INN2: j=0 : A[0] er størst i A[0...0]

2: ***for (j = 0, j < k, j++)*** {

LI2: 0 <= t < j <= n → A[t] <= A[j]

dvs. A[j] er størst i A[0...j]

if (A[j] > A[j+1]) swap(A[j], A[j+1]);

j'=j+1: er A[j'] størst i A[0...j'] ?

hvis A[j] <= A[j+1] & LI2 → LI2'

ellers A[j'] = A[j] > A[j+1] = A[j'–1] & LI2 → LI2'

LI2': j'=j+1: 0 <= t < j' <= n → A[t] <= A[j']

}

UTG2: LI2 & ***j=k*** gir: 0 <= t < k <= n → A[t] <= A[k]

dvs. A[k] er størst bland A[0...k]

UTG2 & LI1 gir: A[0...k–1] <= A[k] <= A[k+1...n] →

LI1': k'= k–1: 1 <= r < s <= n, s > k' → A[r] <= A[s]

dvs. A[k'...n] er sortert og har n-k' største elementer

}

UTG1: **LI1 &** ***k=0*** gir **0 <= r < s <= n & s > 0 → A[r] <= A[s]**

dvs. : A[1...n] er sortert og har n-1 største elementer,
men da har A[0] minste element <= A[r] for 0 < r <= n
(spesielt, for r=1 : A[0] <= A[1])

Oppsummering

- *jdsl.simple.api.Container*
- *jdsl.core.api.Container*
- *java.util.Enumeration*
- *jdsl.simple.api*
  - *Stack*
  - *Queue*
  - *Deque*
  - *RankedSequence*
- *jdsl.core.api*
  - *Position*
  - *PositionalContainer*
  - *PositionalSequence*
  - *Sequence – implementert med*
    - *java.util.Vector*
    - *array*
    - *en-veis liste*
    - *to-veis liste*

- *organisering av ADTer*
  - *hierarki av interface burde avspeile alle relasjoner mellom begrepene*
- *en ADT kan brukes for en "generisk" implementasjon av en annen ADT*
  - *relativ kompleksitet*
- *riktige abstraksjoner kan være vanskelig å finne*
  - *Position*
- *forskjellige implementasjoner av samme ADT*
  - *vurdering av implementasjoner opp mot hverandre*
- *"generisk" algoritme bruker kun ADTer*
  - *grensesnitt informasjon om parametre*
  - *og derfor kan brukes med vilkårlige implementasjoner*
- *sorterings algoritmer*
  - *seleksjon–, merge–, bobble–sortering*